Towards Gradually Typed Hardware Description Languages

Peitian Pan, Shunning Jiang, Yanghui Ou, Christopher Batten
Cornell University
Ithaca, NY, USA

ABSTRACT
Recent research in hardware development methodologies has argued for sophisticated hardware generators and dynamically typed high-level components to improve the hardware design and verification productivity. In this talk, we describe our on-going work towards gradually typed HDLs to realize such productivity benefits through powerful static type checking and safe and performant composition of mixed-typed components.

1 MOTIVATION
Specialized hardware tend to have high non-recurring engineering (NRE) costs that hinder the development of promising hardware systems. Recent research shows that high NRE costs in two ways: (1) parametrized hardware generators to maximize design reuse [1, 3, 10, 12, 16, 18, 20] and (2) dynamically typed high-level components to enable polymorphic test harnesses, reference models, and cycle-approximate models [7, 11–15]. Unfortunately, several challenges have prevented state-of-the-art HDLs from realizing these productivity benefits. First, existing HDLs suffer from a long design-debug cycle because they generally fail to statically type check hardware generators. For example, instead of verifying matching bitwidths statically, most existing HDLs delay bitwidth checks among connections until hardware instances have been generated and bitwidth parameters have been resolved into concrete values. Second, disciplined composition of mixed-typed components is difficult. Statically typed components are guaranteed to drive well-typed values on output ports, as long as all input port values have the expected types. In contrast, dynamically typed components can drive potentially ill-typed values on output ports because the input values can be ill-typed, or there might be a type error in the component. Third, modern dynamically typed HDLs sacrifice simulation performance to ensure a disciplined composition of mixed-typed components. Modern HDLs that support dynamically typed components generally insert simulation-time checks to ensure safe interoperation between components [12]. However, this approach incurs performance overhead and fails to leverage the static type information.

2 A SPECTRUM OF EXISTING HDLs
Table 1 shows that existing HDLs either (1) do not support sophisticated hardware generators and dynamically typed components or (2) have long design-debug cycles due to late type checks.

Traditional HDLs – Verilog/SystemVerilog [9] and VHDL [8] are traditional statically typed HDLs. Traditional HDLs enforce type invariants by type checking the elaborated hierarchy of hardware instances, which happens in the middle of a hardware design cycle.

High-Level Statically Typed HDLs – Bluespec SystemVerilog [17] allows static type checking of a generator to discover potential design issues early in the design cycle. However, Bluespec is not able to detect bitwidth mismatches in vector slicing operations and defers this check to elaboration. C2lash [2] is a Haskell dialect for hardware development. It benefits from Haskell’s static type system and can type check the generators before elaboration.


Embedded Dynamically Typed HDLs – PyRTL [5], MyHDL [6], PyMTL [14], and PyMTL3 [12] are embedded in Python, a dynamically typed programming language. Almost all existing embedded dynamically typed HDLs lack static type checking capabilities, and most of them do not perform elaboration – nor simulation-time type checks. Instead, type errors typically occur when an ill-typed object is used, which causes difficulties for debugging.

Gradually Typed HDLs – GT-HDLs achieve the best of both statically and dynamically typed HDLs. Instead of only allowing either statically or dynamically typed components, GT-HDLs support the safe interoperation of mixed-typed components. Similar to high-level statically typed HDLs, GT-HDLs improve design productivity by performing static type checks on hardware generators to shorten the design-debug cycle. Similar to embedded dynamically typed HDLs, GT-HDLs are flexible and allow dynamically typed high-level components to improve verification productivity. GT-HDLs can also improve verification productivity by improving simulation performance using the available static type information.

3 RESEARCH DIRECTIONS FOR GT-HDLs
We discuss three techniques that address the challenges in §1. We envision that a GT-HDL implementation will incorporate these techniques to boost hardware design and verification productivity.

Static Type Checking for Generators – GT-HDLs need powerful static type checking capabilities to ensure a short design-debug cycle. Existing HDLs either do not perform static checks at all (e.g., embedded dynamically typed HDLs) or require intimate knowledge of advanced type systems (e.g., high-level/embedded statically typed HDLs) that are foreign to most hardware designers.

We propose to build static type checkers that specifically target the HDL syntax of synthesizable hardware generators and translate critical hardware generator properties into integer constraints solvable by a satisfiability modulo theory (SMT) solver. We assume that hardware generators take non-negative integer parameters, which correspond to integer variables in the generated constraints.

For example, a repeater generator generates circuit that duplicates the n-bit input X times to form the output, where n and X are parameters. The proposed static type checker first propagates the symbolic bitwidths to all signals in the generator. It then builds the following bitwidth constraint for the syntax construct that
concatenates the X copies of n-bit input to the output:

\[ n + n + \ldots + n = n \times X \] (1)

If the SMT solver finds \( \neg(1) \) unsatisfiable, the checker has established the proof that both sides of the concatenation has matching bitwidth; if the solver finds a satisfiable assignment to variables \( n \) and \( X \), the checker has found a counterexample that triggers a potential design issue in the generator. The above trivial example can be generalized to verify more properties including bounded array indices and matching bitwidths on slicing operations.

**Safe Mixed-Typed Component Composition** – In response to the second challenge in §1, we propose elaboration-time and simulation-time type checks. Elaboration-time type checks verify the hardware generator parameters against its type signature to prevent admitting ill-typed parameters. Simulation-time type checks are inserted to every signal assignment that crosses the mixed-typed component boundary and verify if LHS and RHS of the assignment have matching bitwidth. Figure 1 shows a composition of a statically typed divider and a dynamically typed test bench. The light blue signals are the targets of simulation-time type checks. Only three simulation-time checks are necessary for correctness because the statically typed domain is guaranteed to generate well-typed operands as long as its input is well-typed.

**Type-Based Simulation Optimizations** – GT-HDLs enable mixed-typed component compositions and can potentially benefit from the static type information of hardware components. To demonstrate how a GT-HDL implementation can leverage static type information for simulation performance, we describe signal coalescing, a technique that reduces the number of signal assignments in simulation. To simulate the behavior of a circuit, the simulator of an HDL generally presents a group of connected signals using the net data structure [12], where one writer signal continuously drives values to zero or more readers. In an unoptimized implementation, the continuous update is generally implemented as per-cycle signal assignments from the writer to all readers, which include a simulation-time type check to prevent a dynamically typed writer from injecting ill-typed values. Instead of performing actual signal assignments, signal coalescing sets up a reference from the group of dynamically typed readers to the dynamically typed writer to eliminate unnecessary assignments. This technique does not undermine the safety of mixed-typed component composition because it still preserves the simulation-time type checks when any dynamically typed signal updates a statically typed signal.

### Table 1: A Spectrum of Existing Hardware Description Languages

<table>
<thead>
<tr>
<th>HDLs</th>
<th>Sophisticated Generators</th>
<th>Dyn. Typied High-Level Components</th>
<th>How are Type Invariants Enforced?</th>
<th>When Type Checks Occur in Design Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traditional HDLs</td>
<td>Verilog/SystemVerilog, VHDL</td>
<td></td>
<td><img src="" alt=" " /></td>
<td><img src="" alt=" " /></td>
</tr>
<tr>
<td>High-Level Stat. Typed HDLs</td>
<td>Bluespec SystemVerilog, C/C++</td>
<td><img src="" alt=" " /></td>
<td><img src="" alt=" " /></td>
<td><img src="" alt=" " /></td>
</tr>
<tr>
<td>Embedded Stat. Typed HDLs</td>
<td>Chisel, SpinalHDL, Lava</td>
<td><img src="" alt=" " /></td>
<td><img src="" alt=" " /></td>
<td><img src="" alt=" " /></td>
</tr>
<tr>
<td>Embedded Dyn. Typied HDLs</td>
<td>PyRTL, MyHDL, PyMTL, PyMTL3</td>
<td><img src="" alt=" " /></td>
<td><img src="" alt=" " /></td>
<td><img src="" alt=" " /></td>
</tr>
<tr>
<td>Embedded Grad. Typied HDLs</td>
<td>GT-HDL*</td>
<td><img src="" alt=" " /></td>
<td><img src="" alt=" " /></td>
<td><img src="" alt=" " /></td>
</tr>
</tbody>
</table>

Sophisticated generators: generators that programatically generate hardware instances. Earlier type checks in the design cycle lead to lower bug-fixing costs. ![ ]( ) ![ ]( ) ![ ]( ) : almost all/some/no invariants enforced; Stats./Dyn./Grad. typed: statically/dynamically/gradually typed; *: our proposal.

**Figure 1: A Mixed-Typed Composition in GT-HDL**

4 **CONCLUSION**

Hardware designers have benefited from the type safety of statically typed HDLs and the flexibility of dynamically typed HDLs. However, existing HDLs remain either statically or dynamically typed, which limits the peak designer productivity. In this paper, we propose GT-HDLs to achieve the best of both worlds. We believe GT-HDLs are a compelling solution to realize the productivity benefits of advances in hardware design methodology research. We hope this paper will spark interests from the methodology research community to tackle the challenges in providing strong static type checking capabilities, disciplined mixed-typed composition, and type-based optimizations to fully realize the potential of GT-HDLs.

**ACKNOWLEDGMENT**

This work was supported in part by the Center for Applications Driving Architectures (ADA), one of six centers of JUMP, a Semiconductor Research Corporation program co-sponsored by DARPA, a research gift from Xilinx, Inc., and donations from Intel.
REFERENCES


